Polymorphic Access Permissions

Nels E. Beckman[*] **Jonathan Aldrich**[†]

March 2010
CMU-ISR-10-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]nbeckman@cs.cmu.edu
[†]jonathan.aldrich@cs.cmu.edu

This work was supported by DARPA Grant #HR00110710019 and a grant from the National Science Foundation, #CCF-0811592. The first author was supported by a National Science Foundation Graduate Research Fellowship, #DGE0234630.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAR 2010		**00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Polymorphic Access Permissions	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This paper presents a polymorphic extension to a type system that prevents the misuse of object protocols. Polymorphism allows classes to be generic in the Access Permissions in their specifications. Access Permissions describe both the current state of an object and whether or not references to the object alias. Polymorphic Access Permissions allow programmers to specify certain patterns that we have encountered in practice, for example a collection of open, unaliased files. This paper also describes an implementation of this system as a static typestate checker for the Java programming language.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**28**	

Keywords: typestate, aliasing, fractional permissions, polymorphism

Abstract

This paper presents a polymorphic extension to a type system that prevents the misuse of object protocols. Polymorphism allows classes to be generic in the Access Permissions in their specifications. Access Permissions describe both the current state of an object and whether or not references to the object alias. Polymorphic Access Permissions allow programmers to specify certain patterns that we have encountered in practice, for example a collection of open, unaliased files. This paper also describes an implementation of this system as a static typestate checker for the Java programming language.

1 Introduction

This paper presents a type system and a related implementation for a language that prevents the misuse of object protocols. Such systems are generally known as typestate checkers. The novel feature of this language is that its type system supports parametric polymorphism over Access Permissions, thereby addressing a key limitation of the type system upon which it directly builds [2]. We are unaware of any other static typestate checker that supports a similar level of expressiveness.

In practice, many classes in object-oriented programs define protocols. The protocols prohibit certain methods of a class from being called at certain times, generally depending on the state of class instances. To take a common example, consider Java's `Iterator` interface. The `next` method on an iterator should only be called if there are actually elements left to iterate over, otherwise the call will result in a run-time exception. While classes may frequently define such protocols, popular languages generally do not check that these protocols are obeyed. As a result, there has been considerable work in the research community on static and dynamic typestate checking [13, 8, 11, 7].

Of the approaches that attempt to check protocol usage statically, the subset that work modularly, checking each method once like a traditional type system, must somehow deal with aliasing. When tracking an object's state as it steps through a method body, the potential for unrestricted aliasing means that any method call might potentially alter the state of that object through hidden references. Typestate checkers typically deal with this problem by imposing aliasing restrictions such as uniqueness [7, 8]. This paper builds on the work of Bierhoff and Aldrich [2] who introduced the notion of Access Permissions for tracking object states and aliasing. Access Permissions associated with program references track not only the state of the object referenced, but the *permission* available to modify that object and potentially held by other references. This permission information is encoded in one of five *permission kinds*, recapped in Figure 1, which greatly increased the flexibility of aliasing when compared with existing approaches.

	Other Refs. May:		
This Ref. May:	No Other Refs.	Read	Write
Read	unique	immutable	pure
Write	unique	full	share

Figure 1: The five permission kinds

Polymoprhism over Access Permissions is important for many of the same reasons that parametric polymorphism is already useful in standard type systems; it increases the precision of the type system for types whose implementations are in some sense "ambivalent" about the objects they reference. In Java 1.5 we can use this feature to define a class `Stack<T>`, a stack that can hold elements of any type. When that class is instantiated with some type, say `File`, we are ensured that that particular instance will only accept and return files.

In our case, Access Permissions statically describe the current state of an object reachable through a reference, and whether or not that reference may be aliased by other references. By

enabling polymorphism over Access Permissions, programmers can write classes that are ambivalent about their elements' protocols and level of aliased-ness. For example, the same `Stack<T>` class can be given polymorphic *specifications*. Later on, a stack can be instantiated in a variety of different ways, say a stack of unique pointers to open files, or a stack of shared pointers to sockets guaranteed to be open.

To give a clearer picture of the difficulties that arise without polymorphism, we will attempt to specify just such a stack using our existing methodology [2]. Figure 2 shows the implementation and specification of a mutable stack. This stack defines only two methods, `push` and `pop`. If `pop` is called when the stack is empty, it simply returns `null`. The stack defines no protocol of any interest, but since it is generic in the types of the elements it holds, its elements very well might.

```
1  @Invariants(@State(name="alive", inv="unique(first) in alive"))
2  class Stack<T> {
3    @Invariants(@State(name="alive",
4      inv="unique(next) in alive * pure(item) in alive"))
5    class Node { T item; Node next; }
6    Node first;
7
8    @Spec(post="unique(this) in alive")
9    Stack() { first = null; }
10
11   @Spec(pre="unique(this) in alive * pure(item) in alive",
12         post="unique(this) in alive")
13   void push(T item)  { Node n = new Node();
14     n.item = item; n.next = first;
15     first = n;
16   }
17
18   @Spec(pre="unique(this) in alive",
19         post="unique(this) in alive * pure(result) in alive")
20   T pop() {
21     if( first == null ) return null;
22     else { T result = first.item;
23       first = first.next; return result;
24     }
25   }
26 }
```

Figure 2: A specification of a `Stack` class, without polymorphism. It is weak in the sense that no matter what state the elements of the stack are in, the caller of `pop` method only knows the element are in the "alive" state.

We have attempted to specify this stack in as general a way as possible. Because the implementation does not constrain the types of the elements it holds, it also does not constrain the protocols defined by those elements. If a programmer only pushes open files on the stack, he expects open files to be returned from the stack. Furthermore, for verification purposes, the implementation does not constrain the permission kind associated with those elements. In other words, any permission

that the caller of the push method is willing to forfeit to the pushed item, can soundly be transferred to the eventual caller of the pop method. For these reasons, the Access Permissions to the stack elements (highlighted in bold) in Figure 2 are as general as possible: the element must be in the "alive" state (trivially satisfied by every object) and the element must have pure permission kind (satisfiable with any other permission kind).

Unfortunately, this specification is quite imprecise. While it is easy to satisfy the pre-condition of the push method (line 11), for any object, the post-condition of the pop method (line 19) is quite weak. For example, for a stack of type Stack<File>, even if we push an open file, and the calling site has the sole reference to that file f (signified by the Access Permission, unique(f) in Open), we lose all this information when calling the pop method. The caller of the pop method receives a permission to read but not modify a file that may be open or closed (pure(result) in Alive). The caller would be unable to use methods that depend on the file being open, for example read. This imprecision is analogous to state of Java collections before generics; the return type of the pop method of Stack could only guarantee that the returned object was of type Object.

In practice [3] we have seen that when using the Access Permissions methodology for typestate checking, one is forced either to copy and re-specify an implementation of a collection several times, once for each context in which it is used, or settle for false-positives. Collections are frequently used as a means of ownership transfer between different threads and different program structures.

This paper presents a type system that allows us to give Stack a single polymorphic specification. A stack of unique, open files can share an implementation with a stack of shared, open sockets without losing precision, such as in the specification of the pop method.

This paper will proceed in the following manner: Section 2 contains a brief recap of the type system of Bierhoff and Aldrich [2] upon which this work directly builds. Section 3 describes the type system in technical detail and contains the primary contribution of this work. As the section progresses, we will attempt to motivate our new features and present a number of useful examples. Since the specifications end up being rather verbose, Section 4 shows how we can introduce syntactic sugar that greatly decreases the size of specifications written by programmers. Section 5 describes our implementation before a discussion of related work and a conclusion.

2 Background: Access Permissions

Because our system is an extension of the type system of Bierhoff and Aldrich [2], some understanding of that system is necessary to understand our work. In this section, we will briefly recap a few of the more novel aspects of this existing work.

Bierhoff and Aldrich [2] presented a type system for statically ensuring that object protocols would not be violated at run-time. This system had a variety of novel features. For one, it allowed a larger variety of aliasing patterns than previous approaches. It is important for static typestate checkers to track aliasing in some manner because of the possibility of other references modifying the state of an object that is being tracked. While existing typestate checkers required linearity for any object with states [7], Bierhoff and Aldrich [2] introduced five permission kinds, described in Figure 1, which record statically which references in a program could be used to modify, and

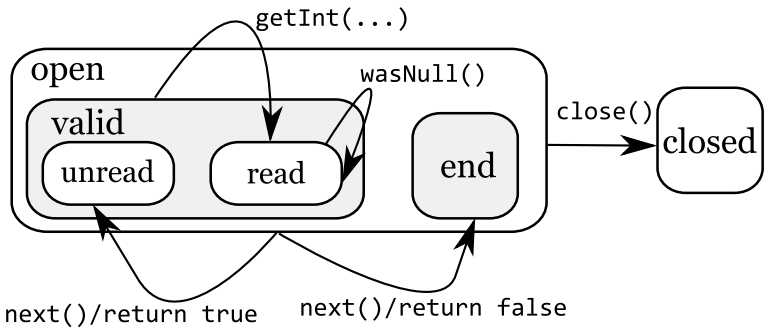

Figure 3: A graphical depiction of the protocol defined by the `ResultSet` class in the Java JDBC library [3].

whether or not other references might alias the same object. The result was greater flexibility of aliasing for tracked objects.

This approach also associated each access permission with a series of fractions. Fractions [6], as used in verification, allow weaker aliasing permissions to be recombined in order to form stronger permissions. The numerical value of the fractions indicate to the analysis at what point it is guaranteed that all aliases have been eliminated. With fractions, a unique permission, for example, can be temporarily split into three share permissions, distributed to references in different parts of the program, and then be recombined into a unique permission.

Finally, this approach allows classes to define state hierarchies, which can exponentially decrease the number of cases that must be considered when writing specifications. If a class defines a number of states, state refinement allows that same class, or its subclasses, to introduce new states inside existing states. Consider the protocol defined by the `ResultSet` class in Java's JDBC library, modeled as a UML State Diagram in Figure 3. At the top level of the state hierarchy defined by this class, there is a simple Open/Closed protocol. When a `ResultSet` is closed by calling the `close` method, any underlying database resources will be released. But within the open state, we can see that there is a much more interesting protocol. Each time the `next` method is called, the database result will advance to the next row in the collection of results. If this method returns false, this indicates that there are no more rows in the result. Only if this method returns true, do we say that the result is in the valid state, at which point the values for each column can be queried. Finally, the `wasNull` method, which checks to see if the last column queried was actually the database value "NULL," can only be called after some column in the current row has been queried. State hierarchies like this one allow simpler specifications, since some methods may be ambivalent as to where exactly in the hierarchy an object lies. For example, the `close` method requires only that the receiver be somewhere in the open state, and does not care if the result is valid, unread, etc.

As part of the state hierarchy methodology, a permission can provide a *state guarantee*. A permission with a guarantee promises that reference cannot be used to leave the guaranteed state. Moreover, because of the means by which guarantees are created, a permission with a guarantee also knows that no other reference can be used to leave that guaranteed state. This mechanism

exists to make certain weak permissions, for example share which indicates that any number of modifying aliases may exist to the same object, much more useful. For instance, we might create a permission for a `ResultSet` that guarantees the open state. This would ensure that no matter how many aliases were created, none of them would ever be able to `close` the object. And thanks to fractions, by gathering back together each permission created with that guarantee, we can remove the guarantee later on, and close the result set.

There is also a notion of state *dimensions*. Dimensions refine states as well, but unlike states which are mutually exclusive (an object can only be in one state at a time on a given level of the hierarchy) dimensions are concurrent (an object must be in one state in every dimension on a given level of the hierarchy). This allows us to model classes that define multiple, orthogonal state machines. Dimensions in practice are quite a bit like Data Groups [12]. We will use the general term "node" to refer to both states and dimensions.

In order to keep track of all of these elements, Access Permissions, the static predicates associated with each reference at every program location, have the following form (see Figure 5 for the full syntax):

$$\mathsf{access}(r, n, g, k, A)$$

r is the reference with which the permission is statically associated. n is the currently guaranteed state. If no state is guaranteed, this will be "alive," the root of the hierarchy for every type. g is the *fraction function*. It maps each state in the state hierarchy between the guaranteed node and "alive" to a fraction between zero and one. By keeping track of these fractions, we can tell when it is safe to remove a previously-established guarantee, because other aliases no longer depend on it. k is the *below* fraction. It exists so the analysis can track whether or not the associated reference has the right to modify the current state of the object (if the value is greater than zero) and if this reference is the only reference that has the ability to do so (if the value is one). Finally, A is the current state of the object pointed to by the reference. In the next section we will show how our polymorphic extension allows programmers to abstract over each element of the access permission.

Access Permissions are tracked linearly as they flow throw the body of a method, and they are updated as each step of the method dictates. By tracking them linearly, we ensure that no permission is unsoundly duplicated. A series of *splitting* and *merging* rules allows permissions of one kind to be created from or in place of other permission kinds where it is sound to do so.

Four of the five permission kinds defined in Figure 1 are actually represented in our calculus as access permissions of this form. These definitions are as follows:

$$\begin{aligned}
&\mathsf{unique} \equiv \mathsf{access}(r, n, \{g, n \mapsto 1\}, 1, A)\\
&\mathsf{full} \equiv \mathsf{access}(r, n, g, 1, A)\\
&\mathsf{share} \equiv \mathsf{access}(r, n, g, k, A) \quad (0 < k < 1)\\
&\mathsf{pure} \equiv \mathsf{access}(r, n, g, 0, A)
\end{aligned}$$

Both unique and full permissions have a below fraction of one, indicating that we can modify A, the current state of the object, through r, and that no other references have that right. The unique permission also has a fraction of one to the guaranteed node n, ensuring that no other references rely on the object being in state n, and giving it the freedom to change the guaranteed state. A share permission has a below fraction between one and zero, indicating that it can be used to

modify the current state of the object, but that other references may also have that right. Finally the pure permission, whose below fraction is zero, does not have the right to modify A.

For simplicity, we will leave the immutable permission kind out of our formal treatment, but it can be represented by adding an additional flag to distinguish share and immutable permissions.

3 Polymorphic Access Permissions

This section describes our type system in technical detail. The basic idea is to take each element of the Access Permission and allow the programmer to abstract over it at the method and class levels. Bounds on these abstracted variables, enforced at instantiation-time, ensure that the well-formedness rules of the Access Permissions are respected. Additionally, and perhaps most interestingly, the system allows programmers to abstract over the *classifiers* of fractions and fraction functions, not just the fractions themselves. This is useful because it allows programmers to instantiate a collection with a permission kind while remaining ambivalent about the exact fraction values.

Figure 4 gives the basic syntax for a core, object-oriented language with mutable state, inspired by Featherweight Java [10]. While this syntax is essentially the same as previous work [2], there are a few points to keep in mind. Each class can declare a number of new states and dimensions, R. Dimensions, d, refine existing states by introducing a number of new, mutually exclusive substates. As previously mentioned, an object must always be in one state in each dimension that it defines. Any state or dimension can be associated with a predicate, called a state invariant, N, that must hold whenever the object is in that node. Our language has a limited form of constructor I which can only specify an object's initial state (or conjunction of states for objects of multiple dimension) and the predicate required to establish that initial state. Field declarations F declare that a field is "mapped" into a node, and that field can only be modified when the object is in that node. This helps ensure that state guarantees actually guarantee an object's concrete state. Every method has a specification, MS, which consists of a pre- and post-condition, showing which permissions it requires for the receiver and parameters, and which permissions it returns upon completion. State invariants and method specifications are written using predicates P (Figure 5), the standard forms of linear logic [9]. Access Permissions p, mention fractions k and fraction functions g. A fraction is a literal 0 or 1, or a fraction divided by two. A fraction function is the mapping of a node to a fraction, a fraction function divided by two or the concatenation of two fraction functions. Programs must be written in a let-normal form. Most terms and expressions are straightforward with the exception of `pack` and `unpack`. These two expressions are used together and delineate portions of code where an object is temporarily not in any state. They help us check that a method actually changes the state of the receiver object when its specification says it does.

3.0.1 The Syntax of Permissions and Abstraction

The most interesting new addition to the syntax is the ability to introduce type variables at the class level, and permission variables at the class and method level. As seen in Figure 4, a class can introduce any number of type variables, β. Type variables allow classes to be generic over

$$
\begin{array}{rrcl}
\textit{programs} & PR & ::= & \langle \overline{CL}, e \rangle \\
\textit{class decl.} & CL & ::= & \texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha : \kappa}]\ \texttt{extends}\ C'\langle\overline{T}\rangle[\overline{a}]\ \{\ \overline{F}\ \overline{R}\ I\ \overline{N}\ \overline{M}\ \} \\
\textit{field decl.} & F & ::= & f : T\ \texttt{in}\ n \\
\textit{state decl.} & R & ::= & d = \overline{s}\ \texttt{refines}\ s_0 \\
\textit{initial state} & I & ::= & \texttt{initially}\ \langle P, s_1 \otimes \ldots \otimes s_n \rangle \\
\textit{state inv.} & N & ::= & n = P \\
\textit{meth. decl.} & M & ::= & T\ m[\overline{\alpha : \kappa}](\overline{T\ x}) : MS = e \\
\textit{meth. spec.} & MS & ::= & P \multimap E \\
\textit{terms} & t & ::= & x \mid \texttt{true} \mid \texttt{false} \\
 & & \mid & t_1\ \texttt{and}\ t_2 \mid t_1\ \texttt{or}\ t_2 \mid \texttt{not}\ t \\
\textit{expressions} & e & ::= & t \mid f \mid \texttt{assign}\ f := t \\
 & & \mid & \texttt{new}\ C\langle\overline{T}\rangle[\overline{a}](\overline{t}) \mid t_0.m[\overline{a}](\overline{t}) \mid \texttt{super}.m[\overline{a}](\overline{t}) \\
 & & \mid & \texttt{if}(t, e_1, e_2) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{unpack}(n, k, A)\ \texttt{in}\ e \mid \texttt{pack}\ n\ \texttt{to}\ A\ \texttt{in}\ e \\
\textit{references} & r & ::= & x \mid f \\
\textit{types} & T & ::= & \texttt{bool} \mid \beta \mid C\langle\overline{T}\rangle[\overline{a}] \\
\textit{nodes} & n & ::= & \alpha \mid s \mid d
\end{array}
$$

classes	C	*fields*	f	*variables*	x		
methods	m	*states*	s	*dimensions*	d	*type variables*	β

Figure 4: Syntax I: Programs, Classes, Terms and Expressions

other types and should be recognizable to those familiar with other polymorphic object calculi, for example FGJ [10][1]. Permission variables are more interesting.

A permission variable, α, can be introduced for the scope of an entire class, or just a method. Each permission variable must be declared with an associated *quantification classifier*, κ, whose syntax is described in Figure 5. This classifier determines what a variable can be used for within its scope, and what sort of permission element can be instantiated for it. Those instantiating elements, a, are applied at the site of the method call and object instantiation expressions and become part of the class types, $C\langle\overline{T}\rangle[\overline{a}]$.

But what is the nature of the quantification classifiers? Recall that an Access Permission, p in Figure 5, has the following form:

$$\mathsf{access}(r, n, g, k, A)$$

Our system allows each element, with the exception of the reference with which the permission is associated, to be abstracted. Therefore, depending on the classifier that is used, a newly introduced variable can stand for n, g, k, or A. The forms of the quantification classifier, κ, therefore are

[1]We have included traditional parametric polymorphism in order to make our examples more compelling. While we have left out more interesting features like F-bounded polymorphism, we believe that these features are orthogonal and can be added without any great difficulty.

$$
\begin{array}{rrcl}
\textit{quant. class.} & \kappa & ::= & \alpha \mid \mathsf{Asmp}(n, \kappa, T) \mid \omega \mid \Omega(\omega, \omega) \\
 & & \mid & \xi \mid \Xi(\xi) \mid \mathsf{Node}_T \\
\textit{fract. funct. type} & \omega & ::= & \mathsf{FF}(n, n, T) \mid \mathsf{UFF}(n, n, T) \\
\textit{fract. type} & \xi & ::= & \mathsf{Fract} \mid \mathsf{Decimal} \mid \mathbf{1} \mid \mathbf{0} \mid \mathsf{LessThan1} \mid \mathsf{GreaterThan0} \\
\textit{inst. elems.} & a & ::= & \alpha \mid A \mid \omega \mid \xi \mid k \mid g \\
\textit{permissions} & p & ::= & \mathsf{access}(r, n, g, k, A) \\
\textit{facts} & q & ::= & t = \mathtt{true} \mid t = \mathtt{false} \\
\textit{assumptions} & A & ::= & \alpha \mid n \mid A_1 \otimes A_2 \mid A_1 \oplus A_2 \\
\textit{fraction fct.} & g & ::= & \alpha \mid n \mapsto k \mid g/2 \mid g_1, g_2 \\
\textit{fractions} & k & ::= & \alpha \mid 1 \mid 0 \mid k/2 \\
\textit{predicates} & P & ::= & p \mid q \mid P_1 \otimes P_2 \mid \mathbf{1} \mid P_1 \,\&\, P_2 \mid \top \mid P_1 \oplus P_2 \mid \mathbf{0} \\
 & & \mid & \exists \alpha{:}\kappa.P \\
\textit{expr. types} & E & ::= & \exists x : T.P \\
\textit{fract. terms} & h & ::= & g \mid k \\
\textit{valid context} & \Gamma & ::= & \cdot \mid \Gamma, \textit{CL} \mid \Gamma, x{:}T \mid \Gamma, \beta \mid \Gamma, \alpha{:}\kappa \mid \Gamma, q \\
\textit{linear context} & \Delta & ::= & \cdot \mid \Delta, P \\
\textit{packedness} & \upsilon & ::= & \bullet \mid \mathsf{unpacked}(n, g, k, A)
\end{array}
$$

$$\textit{quantification variables} \quad \alpha$$

Figure 5: Syntax II: Permissions, Abstraction and Checking

Node, a node type, ω, a fraction function type, ξ, a fraction type, and Asmp, an assumption type, respectively. Variables of type ω can only be instantiated with fraction functions, and variables of type ξ can only be instantiated with fractions, etc. The fact that these newly introduced quantification variables can be used as elements of the Access Permission is reflected in the syntax, as α appears as a valid form of the syntactic categories n, g, k, and A.

The three other forms of quantification classifiers, α, Ω, and Ξ, are used to further abstract over the classifiers themselves, "one level up." They will be covered in a subsequent section. Thus far we have also neglected to discuss the various adornments of the quantification classifiers, such as n, κ and T in the classifier $\mathsf{Asmp}(n, \kappa, T)$. These adornments form an overall part of the bound on the quantification variable, and as we will show in the next section, are necessary in order to ensure that Access Permissions that mention quantification variables remain well-formed.

3.0.2 The Static Semantics of Permissions and Abstraction

Every time a programmer writes down a specification, which may consist of a number of Access Permissions, the static semantics of our language ensure that those permissions are well-formed. The system's well-formedness rules prevent certain programmer mistakes, such as the use of abstract states that have not been defined. These well-formedness rules motivate many of the features of our quantification classifiers. Let us consider the permission well-formedness rule presented by

Bierhoff [1]:

OLD WF-PERM

$$\frac{\Gamma \vdash g : \overline{n} \mapsto \mathsf{Fract} \qquad \Gamma \vdash r : C \qquad C \vdash A \prec n \qquad \overline{n} = \{\textit{all nodes between } \mathsf{alive} \textit{ and } n \textit{ inclusive for } C\} \qquad \Gamma \vdash k : \mathsf{Fract}}{\Gamma \vdash \mathsf{access}(r, n, g, k, A)\ \mathsf{wf}}$$

This is to say that, in some type-checking context, a permission is well-formed if the reference has class type C, the assumption A only mentions nodes below or equal to the guaranteed node n in the state hierarchy of C, the fraction function g maps a sequence of nodes $\overline{n}$ to fractions, those nodes include all of nodes of C between alive and n, and k is a fraction. Since at the time that quantification variables are introduced it is not known exactly which permission elements will be instantiated for them, it is the job of the quantification classifiers to ensure that a well-formed permission mentioning quantification variables will remain well-formed when those variables are instantiated.

Suppose we wanted to create a simple class that holds a field of parametrized type and with parametrized permission. Here is how we might declare such a class:

```
class OneField<β>[α_n : Node_β, α_g : FF(α_n, alive, β), α_k : Fract, α_A : Asmp(α_n, Fract, β)]
    extends Object <>[] {
  f : β in alive
  ...
  alive = access(f, α_n, α_g, α_k, α_A) // State invariant
  ...
}
```

Let us examine each classifier in turn. α_n, an abstraction of a guaranteed node, is declared to have the classifier Node_β. This classifier says that α_n must be instantiated with a node, and that node must be a node in the state hierarchy of type β. While we do not, as of yet, know what this type will be, α_n will be instantiated after the type variable β, at which point it will be clear whether or not the instantiated node is a node of the instantiated type. Next, α_g is classified as $\mathsf{FF}(\alpha_n, \mathsf{alive}, \beta)$. This tells us that α_g can only be instantiated with fraction functions, and those fraction functions must contain a fraction for every node in the state hierarchy of β between α_n and alive inclusive. Note how the bound of one quantification variable is dependent on other quantification variables. The classifier of α_k, Fract says that it can only be instantiated with a fraction. Finally, the classifier for α_A, $\mathsf{Asmp}(\alpha_n, \mathsf{Fract}, \beta)$ records that the variable can only be instantiated with assumptions (i.e., the syntactic form A). Furthermore, it stipulates that the instantiating assumption must be below α_n in the state hierarchy of type β and, if the classifier Fract can classify fractions below one (which in this case is trivially true!) the instantiating element for α_A must be equal to α_n.

This last restriction deserves some mention. If a collection holds elements of share or pure permission kind, it must account for the fact that the state of these elements can be changed under the guaranteed node at any time. The assumption is therefore tied to the guarantee and can only be below the guarantee if the eventual instantiating fraction for α_k is one. (At the moment, α_A must trivially always be equal to α_N, but after "classifier classifiers" are introduced, this will no longer be the case.)

$$\textsc{Var}\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \textsc{SubSm}\ \frac{\Gamma \vdash a : \kappa \qquad \Gamma \vdash \kappa \sqsubseteq \kappa'}{\Gamma \vdash a : \kappa'} \qquad \Gamma \vdash \omega : \Omega(\omega, \omega)$$

$$\Gamma \vdash \xi : \Xi(\xi) \qquad \Gamma \vdash 0 : \mathbf{0} \qquad \Gamma \vdash 1 : \mathbf{1} \qquad \frac{\Gamma \vdash k : \mathsf{GreaterThan0}}{\Gamma \vdash k/2 : \mathsf{Decimal}}$$

$$\textsc{Same Fract}\ \frac{\Gamma \vdash k : \kappa \qquad \kappa \sqsubseteq \mathsf{LessThan1}}{\Gamma \vdash k/2 : \kappa} \qquad \frac{\Gamma \vdash k : \mathsf{Fract}}{\Gamma \vdash k/2 : \mathsf{Fract}} \qquad \textsc{FF-Div2}\ \frac{\Gamma \vdash g : \mathsf{FF}(n_1, n_2, T)}{\Gamma \vdash g/2 : \mathsf{FF}(n_1, n_2, T)}$$

$$\frac{\Gamma \vdash g : \mathsf{UFF}(n_1, n_2, T)}{\Gamma \vdash g/2 : \mathsf{FF}(n_1, n_2, T)} \qquad \frac{\Gamma \vdash k : \mathsf{Decimal} \qquad \Gamma \vdash n : \mathsf{Node}_T}{\Gamma \vdash n \mapsto k : \mathsf{FF}(n, n, T)}$$

$$\frac{\Gamma \vdash k : 1 \qquad \Gamma \vdash n : \mathsf{Node}_T}{\Gamma \vdash n \mapsto k : \mathsf{UFF}(n, n, T)} \qquad \frac{\Gamma \vdash g_1 : \mathsf{UFF}(n, n', T) \qquad \Gamma \vdash g_2 : \mathsf{FF}(n', n'', T)}{\Gamma \vdash g_1, g_2 : \mathsf{UFF}(n, n'', T)}$$

$$\frac{\Gamma \vdash g_1 : \mathsf{FF}(n, n', T) \qquad \Gamma \vdash g_2 : \mathsf{FF}(n', n'', T)}{\Gamma \vdash g_1, g_2 : \mathsf{FF}(n, n'', T)}$$

Figure 6: Classification of fractions and fraction functions.

The responsibility of ensuring that an instantiating permission element, a, satisfies the bound imposed on it by a quantification classifier, κ falls on our type system. This is accomplished with the judgment, $\Gamma \vdash a : \kappa$, which says that under a valid typing context Γ, the instantiating element a can be classified with κ. The rules for this judgment are shown in Figures 6 and 7.

Some discussion of these rules is in order. The VAR rule says that any quantification variable has the classifier that it was declared to have. The SUBSM rule says that any element a with classifier κ can be treated as being of classification κ' if κ is a subclassifier of κ'. The next two rules say that classifiers themselves have classifiers, which we will motivate later. Every fraction form has a classifier, including the literals 1 and 0, whose classifiers are the literals themselves. Fraction functions can be classified as either FF, the classification of all fraction functions, or as UFF, the classification of *unique* fraction functions, that is fraction functions whose lowest node maps to the fraction 1. Rule ALIVE says that alive is a node for any type. Rule GROUND says that a node is defined in class C at any instantiation if it is declared in class C. Finally, any node n can be an assumption below or equal to node n for any fraction classifier, but two assumptions can only be joined to form an assumption if both assumptions are below some common node n, and the classifier bound in Asmp is $\mathbf{1}$.

Now that we have seen the variety of classifiers available in our type system and how each permission element is classified, let us present the new well-formedness rule for permissions, which

$$\textsc{Alive}\quad \Gamma \vdash \mathsf{alive} : \mathsf{Node}_T \qquad \textsc{Ground}\quad \frac{\Gamma \vdash n \ \mathsf{from}\ C}{\Gamma \vdash n : \mathsf{Node}_{C\langle\overline{\beta}\rangle[\overline{a}]}} \qquad \frac{\Gamma \vdash n : \mathsf{Node}_T}{\Gamma \vdash n : \mathsf{Asmp}(n, \kappa, T)}$$

$$\frac{\Gamma \vdash A_1 : \mathsf{Asmp}(n', \mathbf{1}, T) \qquad \Gamma \vdash A_2 : \mathsf{Asmp}(n'', \mathbf{1}, T) \qquad \Gamma \vdash n' \leq n \qquad \Gamma \vdash n'' \leq n}{\Gamma \vdash A_1 \otimes A_2 : \mathsf{Asmp}(n, \mathbf{1}, T)}$$

$$\frac{\Gamma \vdash A_1 : \mathsf{Asmp}(n', \mathbf{1}, T) \qquad \Gamma \vdash A_2 : \mathsf{Asmp}(n'', \mathbf{1}, T) \qquad \Gamma \vdash n' \leq n \qquad \Gamma \vdash n'' \leq n}{\Gamma \vdash A_1 \oplus A_2 : \mathsf{Asmp}(n, \mathbf{1}, T)}$$

Figure 7: Classification of nodes and assumptions.

updates OLD WF-PERM presented earlier in this section:

$$\text{WF-Perm}\ \frac{\begin{array}{ccc}\Gamma \vdash r : T & \Gamma \vdash n : \mathsf{Node}_T & \\ \Gamma \vdash g : \mathsf{FF}(n, \mathsf{alive}, T) & \Gamma \vdash k : \kappa & \Gamma \vdash \kappa \sqsubseteq \mathsf{Fract} \qquad \Gamma \vdash A : \mathsf{Asmp}(n, \kappa, T)\end{array}}{\Gamma \vdash \mathsf{access}(r, n, g, k, A)\ \mathsf{wf}}$$

Thanks to our changes, the classifiers of each element of the permission succinctly express the restrictions on each element. Note that the classifier of k, κ is the same κ mentioned in A's classifier. This restriction, coupled with the assumption classification rules in Figure 7, ensure that $n = A$ for any polymorphic permission with a fraction k less than one. Using WF-PERM, our type system would find that the state invariant for the alive state in the `OneField` class is indeed well-formed:

$$\beta, f : \beta, \alpha_n : \mathsf{Node}_\beta, \alpha_g : \mathsf{FF}(\alpha_n, \mathsf{alive}, \beta), \alpha_k : \mathsf{Fract}, \alpha_A : \mathsf{Asmp}(\alpha_n, \mathsf{Fract}, \beta) \vdash \mathsf{access}(f, \alpha_n, \alpha_g, \alpha_k, \alpha_A)\ \mathsf{wf}$$

The quantification classifiers also form a number of interesting subclassification relationships. Subclassification allows programmers to write specifications that are quite expressive, in a way that is analogous to Java's F-bounded polymorphism. Subclassification is established with the judgment $\Gamma \vdash \kappa \sqsubseteq \kappa$. The rules for this judgment are presented in Figure 8.

The main points of interest are the relationships between fraction classifiers, and the relationships between fraction function classifiers. Fraction classifiers form a hierarchy from Fract, the classifier of every fraction, to 0, 1, and Decimal, the classifiers for 0, 1, and fractions between 0 and 1, respectively. LessThan1 and GreaterThan0 have the obvious locations in this hierarchy. Fraction functions can be classified by FF, or its subclassifier UFF. Fraction functions classified by UFF have their lowest node mapped to 1, and are the fraction functions used for unique permissions.

3.0.3 Abstracting Over Quantification Classifiers

While the ability to abstract over fractions and fraction functions is useful, it is not quite as flexible as we would like. Consider the following scenario: We would like to take our stack, presented

REFLEXIVE
$$\Gamma \vdash \kappa \sqsubseteq \kappa$$

TRANSITIVE
$$\frac{\Gamma \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \vdash \kappa' \sqsubseteq \kappa''}{\Gamma \vdash \kappa \sqsubseteq \kappa''}$$

$$\Gamma \vdash 1 \sqsubseteq \mathsf{GreaterThan0}$$

$$\Gamma \vdash \mathsf{Decimal} \sqsubseteq \mathsf{GreaterThan0}$$

$$\Gamma \vdash \mathsf{GreaterThan0} \sqsubseteq \mathsf{Fract}$$

$$\Gamma \vdash 0 \sqsubseteq \mathsf{LessThan1}$$

$$\Gamma \vdash \mathsf{Decimal} \sqsubseteq \mathsf{LessThan1}$$

$$\Gamma \vdash \mathsf{LessThan1} \sqsubseteq \mathsf{Fract}$$

$$\Gamma \vdash \mathsf{UFF}(n_1, n_2, T) \sqsubseteq \mathsf{FF}(n_1, n_2, T)$$

$$\frac{\Gamma \vdash n \leq n' \text{ in } T \qquad \Gamma \vdash \kappa \sqsubseteq \kappa'}{\Gamma \vdash \mathsf{Asmp}(n, \kappa, T) \sqsubseteq \mathsf{Asmp}(n', \kappa', T)}$$

$$\frac{\Gamma \vdash \omega_1' \sqsubseteq \omega_1 \qquad \Gamma \vdash \omega_2 \sqsubseteq \omega_2'}{\Gamma \vdash \Omega(\omega_1, \omega_2) \sqsubseteq \Omega(\omega_1', \omega_2')}$$

$$\frac{\Gamma \vdash \xi \sqsubseteq \xi'}{\Gamma \vdash \Xi(\xi) \sqsubseteq \Xi(\xi')}$$

FF UPPER-BOUND
$$\frac{\Gamma \vdash \alpha : \Omega(\_, \omega)}{\Gamma \vdash \alpha \sqsubseteq \omega}$$

FF LOWER-BOUND
$$\frac{\Gamma \vdash \alpha : \Omega(\omega, \_)}{\Gamma \vdash \omega \sqsubseteq \alpha}$$

FRACT UPPER-BOUND
$$\frac{\Gamma \vdash \alpha : \Xi(\xi)}{\Gamma \vdash \alpha \sqsubseteq \xi}$$

Figure 8: Subclassification rules

back in Section 1, and specify it is generic over the permission kind of the elements it holds, but where every each element must be of the same kind. We will only concentrate on the class quantification variables and the `push` method, since this will be enough to motivate higher quantification. Consider the following specification of `Stack`:

```
class Stack<β>[α_n : Node_β, α_g : FF(α_n, alive, β), α_k : Fract, α_A : Asmp(α_n, Fract, β)]
  extends Object <>[] { ...
  boolean push(T i) : unique(this) ⊗ access(i, α_n, α_g, α_k, α_A) ⊸ unique(this)
  ... }
```

Now, suppose that at a particular instantiation site, we would like to use this stack, and we would like it instantiated as a stack of shared permissions to files that are guaranteed to be open. What instantiations should we use? Unfortunately, we are required to choose definite values for the fraction α_k, and the fraction function, α_g. Let us assume that we instantiate the stack as follows; $\mathtt{Stack}\langle\mathtt{File}\rangle[\mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{2}, \mathsf{Open} \mapsto \frac{1}{2}\}, \frac{1}{2}, \mathsf{Open}]$. This means that in an environment where the permission $\mathsf{access}(r_1, \mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{2}, \mathsf{Open} \mapsto \frac{1}{2}\}, \frac{1}{2}, \mathsf{Open})$, a share permission, is available for r_1, the call $\mathtt{push}(r_1)$ is legal. Unfortunately, if we have another share permission with different fraction values, say $\mathsf{access}(r_2, \mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{4}, \mathsf{Open} \mapsto \frac{1}{4}\}, \frac{1}{4}, \mathsf{Open})$, the call $\mathtt{push}(r_2)$ is *not* legal, because the pre-condition for the `push` method when instantiated is $\mathsf{unique}(\mathtt{stack}) \otimes \mathsf{access}(r_1, \mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{2}, \mathsf{Open} \mapsto \frac{1}{2}\}, \frac{1}{2}, \mathsf{Open})$. This requires the *exact* same fraction and fraction function values.

To accomplish our original goal of instantiating a stack that can hold share permissions at any fraction, we need more power in the specification language. We need the ability to quantify over the classifiers themselves. Fortunately, the quantification classifiers Ω and Ξ let us do exactly that. Ω is the classifier of all fraction function classifiers. It stores an upper bound and a lower bound of the classifiers that can legally be used to instantiate it. Ξ is the classifier of fraction classifiers. It stores an upper bound of the classifiers that can legally be used to instantiate it. (Why no lower bound? It was not found to be useful for any of our examples. Adding it would be fairly straightforward.) By abstracting over these "classifier classifiers," we can specify that certain fractions and fraction functions must be similar but not identical.

With Ω and Ξ at our disposal, we can finally specify the `Stack` class in the way that we desire:

```
class Stack<β>[αn : Node_β, αω : Ω(UFF(αn, alive, β), FF(αn, alive, β)),
 αξ : Ξ(Fract), αA : Asmp(αn, αξ, β)]
  extends Object <>[] {
  ...
  boolean push(T i) :
     unique(this) ⊗ (∃αg:αω.∃αk:αξ.access(i, αn, αg, αk, αA)) ⊸ unique(this)
  ... }
```

With a stack instantiated as, `Stack`⟨File⟩[Open, FF(Open, alive, `File`), Decimal, Open], we can call the `push` method and pass share permissions of any fractional value. This is in part thanks to the existential quantification that has been added to the `push` method's specification. When instantiated, it can accept any fraction as long as that fraction is classified by Decimal, and any fraction function, provided it is classified by FF.

3.0.4 Quantifying Over Symmetric Permission Kinds

Up until this point, we have used `Stack` as a running example. One of the notable features of stack is that it can hold permissions of any kind. This is largely due to its implementation. A programmer can push an object, and the stack will capture some permission associated with that object. Later on, when the `pop` method is called, the entire permission to the returned element is forfeited by the stack. This means that no matter what permission kind the stack holds, we can count on getting it back later in the execution.

However, some data structures do not provide this feature, and yet could still reasonably support multiple permission kinds. The polymorphic type system we have presented here allows us to precisely specify the behavior of these classes. Consider the mutable linked list class shown in Figure 9. It, like many of the collection classes in the Java standard library, provides random access to its elements. If we would like to use this list in a larger program that we are attempting to verify, we must ask what kind of permission we can get back from the `get` method, especially in light of multiple calls to the same element:

```
Object o_1 = list.get(0);
Object o_2 = list.get(0);
```

Does the second call return the same permission? Does it return no permission? Does it generate an error? There are multiple ways we might want our list to behave. One observation is that this linked list can hold elements of any permission kind that can be split indefinitely to produce the same

```
1  class LinkedList<T> {
2    class Node {  T item; Node next;
3
4      T get(int i, int cur) {
5        if( i == cur ) return item;
6        else return next == null ? null :
7             next.get(i, cur + 1);
8      }
9    }
10
11   int size = 0; Node first = null;
12
13   int size() {...}  void add(T item) {...}
14
15   T get(int i) {
16     if( first == null ) return null;
17     else return first.get(i,0);
18   }
19 }
```

Figure 9: A linked list that provides random access to its elements.

permission. We call such permissions "symmetric," and both the share and pure permissions have this property (along with the immutable permission, which is not part of our formal treatment). Using classifier bounds, our type system allows us to specify `LinkedList` in such a way that it can be used for share and pure but not full or unique.

Here is how we might specify the `LinkedList` class: First, we will introduce bounded quantifiers at the class level:

```
class LinkedList<β>[α_n : Node_β, α_ω : Ω(FF(α_n, alive, β), FF(α_n, alive, β)),
 α_ξ : Ξ(LessThan1), α_A : Asmp(α_n, α_ξ, β)] {
...
}
```

Here note that the fraction classifier α_ξ is bounded so that it can never classify any fraction whose value is 1 (which would be necessary for a unique or full permission). The fraction function classifier α_ω is bounded from below by FF, which means that it can never be used to classify a unique fraction function (i.e., the fraction function that would be used in a unique permission).

The effect of these bounds are two-fold. First, they prevent unique and full permissions from ever being used to instantiate the linked list. This generally means that a unique or full permission cannot be returned as a result of calling the `get` method, although because of splitting these permissions could still be used to satisfy the pre-condition of the `add` method. Secondly, these bounds give the analysis enough information to know internally that fractions classified by α_ξ and fraction functions classified by α_ω can be split and still result in a fraction of the same classification. Rules SAME FRACT and FF-DIV2 in Figure 6 make this possible.

To better illustrate this idea, let us attempt to verify an implementation of the `get` method of the `Node` class, beginning on line 4 of Figure 9. Here is a specification along with an implementation,

assuming the quantified variables introduced in the previous listing are in scope, and that our language allows us to work with integers:

```
class Node {
  alive = unique(next) ⊗ (∃α_g:α_ω.∃α_k:α_ξ.access(result, α_n, α_g, α_k, α_A))
  ...
  β get(int i, int cur) :
  unique(this) ⊸ (∃α_g:α_ω.∃α_k:α_ξ.access(result, α_n, α_g, α_k, α_A)) ⊗ unique(this) {
    if( i == cur,
        unpack(alive,1,alive) in
        let r = item in
        pack alive to alive in r,
      let n = next in
      if( n == null, null, n.get(i,cur+1) )
  }
  ...
}
```

Verifying the `get` method requires proving the permission $\exists\alpha_g{:}\alpha_\omega.\ \exists\alpha_k{:}\alpha_\xi.\ \mathsf{access}(\ r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$ twice, once to satisfy the post-condition, and once to enable the receiver to be packed to the alive state. While splitting rules essentially always allow a access permission to be split in two, by dividing its fraction and fraction functions in half, it is the bounds on the classification variables that ensure the divided fraction and fraction function are still classified by α_ξ and α_ω.

Rules SAME FRACT and FF-DIV2 allow the following verification condition to succeed:

$$\alpha_\omega : \Omega(\ldots), \alpha_g : \alpha_\omega, \alpha_\xi : \Xi(\ldots), \alpha_k : \alpha_\xi; \mathsf{access}(\mathit{result}, \alpha_n, \alpha_g/2, \alpha_k/2, \alpha_A) \vdash (\exists\alpha_g{:}\alpha_\omega.\exists\alpha_k{:}\alpha_\xi.\mathsf{access}(\mathit{result}, \alpha_n, \alpha_g, \alpha_k, \alpha_A))$$

3.0.5 Typing Rules

The type-checking rules for expressions in our language are largely similar to the ones presented by Bierhoff and Aldrich [2]. Unfortunately, due to space constraints, we are unable to present them all here. Instead, Figure 10 presents only the rules that have changed due to polymorphism and for the remainder we refer the interested reader to existing work. The main typing judgment is $\Gamma; \Delta; \upsilon \vdash_C e : E; \upsilon'$, which means, in the context of some valid facts Γ, some linear facts Δ, some packed-ness state υ, and within the context of class C, the expression e has type E, and will finish in the packed-ness state υ'.

Our type system allows abstract states of an object to be associated with predicates over the fields of the object which must hold whenever an object is in the abstract state. We refer to these predicates as "state invariants." We use a packing/unpacking methodology [8] in order check that state invariants do hold, even in the face of reentrant objects. In this methodology, a programmer will exchange a permission to the receiver in a certain state for the state invariant predicate by using the unpack[2] expression. Before the end of a method body, and before each method call that is potentially reentrant, a programmer must use the pack expression to reacquire permission to the receiver, at which point there is the burden of proving the state invariant for the state to which it

[2]Note that in Plural, our implementation of this approach, pack and unpack operations are inferred [3].

P-NEW

$$\frac{\Gamma \vdash C\langle\overline{T_0}\rangle[\overline{a}]\ \mathsf{wf} \qquad \Gamma \vdash \mathsf{init}(C\langle\overline{T_0}\rangle[\overline{a}]) = \langle\overline{f:T}, \overline{\alpha:\kappa}, P, A\rangle \qquad \Gamma \vdash \overline{t:T} \qquad \Gamma \vdash \overline{a:\kappa} \qquad \Gamma;\Delta \vdash [\overline{t/f}]P}{\Gamma;\Delta;\upsilon \vdash_C \ \mathtt{new}\ C\langle\overline{T_0}\rangle[\overline{a}](\overline{t}) : \exists x{:}C\langle\overline{T_0}\rangle.\mathsf{access}(x, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, 1, A);\upsilon}$$

P-CALL

$$\frac{\Gamma \vdash t_0 : C\langle\overline{T_0}\rangle[\overline{a_0}] \qquad \Gamma \vdash \mathsf{sargs}(m, C\langle\overline{T_0}\rangle[\overline{a_0}]) = (\overline{\alpha:\kappa}) \qquad \Gamma \vdash \overline{a:\kappa} \qquad \Gamma \vdash \mathsf{mtype}(m[\overline{a}], C\langle\overline{T_0}\rangle[\overline{a_0}]) = (\overline{x:T}, P \multimap E) \qquad \Gamma \vdash \overline{t:T} \qquad \Gamma;\Delta \vdash [t_0/\mathtt{this}][t_0/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P}{\Gamma;\Delta;\bullet \vdash_C t_0.m[\overline{a}](\overline{t}) : [t_0/\mathtt{this}][\overline{t}/\overline{x}]E;\bullet}$$

P-SUPER

$$\frac{\Gamma \vdash \mathtt{this} : C\langle\overline{T_t}\rangle[\overline{a_t}] \qquad \Gamma \vdash \mathsf{stype}(C\langle\overline{T_t}\rangle[\overline{a_t}]) = C'\langle\overline{T_s}\rangle[\overline{a_s}] \qquad \Gamma \vdash \mathsf{sargs}(m, C'\langle\overline{T_s}\rangle[\overline{a_s}]) = (\overline{\alpha:\kappa}) \qquad \Gamma \vdash \overline{a:\kappa} \qquad \Gamma \vdash \mathsf{mtype}(m[\overline{a}], C'\langle\overline{T_s}\rangle[\overline{a_s}]) = (\overline{x:T}, P \multimap E) \qquad \Gamma \vdash \overline{t:T} \qquad \Gamma;\Delta \vdash [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P}{\Gamma;\Delta;\bullet \vdash_C \ \mathtt{super}.m[\overline{a}](\overline{t}) : [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E;\bullet}$$

Figure 10: Expression typing rules modified due to polymorphism

is packed. The context υ tracks whether or not the current receiver is packed or unpacked (our language only allows access to fields of the current method receiver).

Several typing rules defer to a linear logic proof judgment, $\Gamma;\Delta \vdash P$ from [2]. We use Linear Logic [9] to ensure that Access Permissions, which are descriptions of how objects are aliased, are not duplicated in an unsound manner. The judgment says that in the context of some valid facts Γ and some linear facts Δ, the predicate P is true.

Rule P-NEW checks an instantiation expression. After checking that the instantiated type is well-formed, the init function takes an instantiated class type and returns the types of its fields, the classifications of the polymorphic variables, the initial object state A and the state invariants for that state P. Both the types of the fields and the initial state invariant are returned in terms of the *instantiating* types and permission elements, as the definition of the init function in Figure 11 explains. The rule then checks that the instantiating elements $\overline{a}$ are actually classified by $\overline{\kappa}$ and then uses the current linear context to prove the required permissions P, but for the arguments that are passed to the constructor, rather than the fields.

The rule P-CALL checks a method call site. The receiver is checked to ensure that it has some kind of class type. The sargs function, defined in Figure 11, looks up the classifiers of the static function parameters, and then the permission arguments, $\overline{a}$, are checked to ensure they have the same classifiers. Additionally, the method arguments are checked to ensure that they have the same types as the method parameters. Note that the mtype function (Figure 11) takes into account the static arguments of t_0's type, $C\langle\overline{T_0}\rangle[\overline{a_0}]$. Finally, the linear context is used to prove the method pre-condition, after all of the appropriate substitutions are made. The rule for type-checking calls

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T'}\rangle[\overline{a'}]\{\ldots\ \overline{f:T\ \texttt{in}\ n}\ \texttt{initially}\langle P, s_1 \otimes \ldots \otimes s_n\rangle\ldots\} \in \Gamma \quad \Gamma \vdash \mathsf{init}(C'\langle\overline{T'}\rangle[\overline{a'}]) = (\overline{f'':T''}, \overline{\alpha':\kappa'}, P', A') \quad \Gamma; (P, \mathsf{access}(\texttt{super}, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, A')) \vdash \mathsf{inv}_C(\mathsf{alive}, A) \otimes \top}{\Gamma \vdash \mathsf{init}(C\langle\overline{T}\rangle[\overline{a}]) = ([\overline{T}/\overline{\beta}][\overline{a}/\overline{\alpha}](\overline{f:T}, \overline{f'':T''}), [\overline{a}/\overline{\alpha}](P \otimes P'), A)}$$

$$\Gamma \vdash \mathsf{init}(\texttt{Object}\langle\rangle[]) = (\cdot, \cdot, 1, \mathsf{alive})$$

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T_s}\rangle[\overline{a_s}]\{\ldots\overline{M}\ldots\} \quad T\ m[\overline{\alpha_m:\kappa_m}](\overline{T\ x}) : P \multimap E = e \in \overline{M} \qquad \overline{\kappa'_m} = [\overline{a}/\overline{\alpha}]\overline{\kappa_m}}{\Gamma \vdash \mathsf{sargs}(m, C\langle\overline{T}\rangle[\overline{a}]) = (\overline{\alpha_m:\kappa'_m})}$$

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T_s}\rangle[\overline{a_s}]\{\ldots\overline{M}\ldots\} \quad T\ m[\overline{\alpha_m:\kappa_m}](\overline{T\ x}) : P \multimap E = e \in \overline{M} \quad \overline{T'} = [\overline{T_c}/\overline{\beta}]\overline{T} \qquad P' = ([\overline{a}/\overline{\alpha_m}]([\overline{a_c}/\overline{\alpha}]P)) \qquad E' = [\overline{T_c}/\overline{\beta}]([\overline{a}/\overline{\alpha_m}]([\overline{a_c}/\overline{\alpha}]E))}{\Gamma \vdash \mathsf{mtype}(m[\overline{a}], C\langle\overline{T_c}\rangle[\overline{a_c}]) = (\overline{x:T'}, P' \multimap E')}$$

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T_s}\rangle[\overline{a_s}]\{\ldots\} \in \Gamma \qquad \overline{T'} = [\overline{T}/\overline{\beta}]T_s \qquad \overline{a'} = [\overline{a}/\overline{\alpha}]a_s}{\Gamma \vdash \mathsf{stype}(C\langle\overline{T}\rangle[\overline{a}]) = C'\langle\overline{T'}\rangle[\overline{a'}]}$$

Figure 11: Various utility judgments used by type-checking and well-formedness rules.

of superclass methods, P-SUPER, works very much in the same way. Our type system uses a "frames" methodology [8] for ensuring soundness in the face of subclassing, here evident in the appearance of the $\texttt{this}_{\mathsf{fr}}$ and `super` references.

Beyond the expression typing rules, there are also a number of rules for ensuring that an entire program is well-formed. These are given in Figure 12. Rule P-CLASS checks that a class is well-formed by adding all of the fields, type variables and quantification variables to the valid context. Every declared quantification classifier is checked to ensure that it is well-formed. It then checks that the field, state, method, constructor and state invariant declarations are well-formed. Rule P-METHOD checks that an method's body correctly implements its specification. First, the classification quantifiers and argument types are checked for well-formedness. An augmented context is used to check that the pre- and post-conditions are well-formed. The override judgment checks that the method's specification is behaviorally compatible with any methods it overrides. Finally, given the permissions specified in the post-condition, the method body is type-checked to ensure that it correctly satisfies its post-condition, and that the receiver is packed on return from the method. Again, we regret that due to space constraints, some judgments that are reused without changes from existing work, such as override, inv_C, linear logic proof, permission splitting and joining rules, and the field, state and constructor well-formedness judgments, are not presented in this paper. Interested readers are referred to existing work [1].

P-CLASS

$$\frac{\begin{array}{c}\mathsf{ftypes}(\overline{F}) = \overline{f:T} \qquad \Gamma' = \Gamma, \overline{\beta}, \overline{f:T}, \overline{a:\kappa} \qquad \Gamma' \vdash \overline{\kappa\ \mathsf{wf}} \qquad \Gamma' \vdash C'\langle\overline{T}\rangle[\overline{a}]\ \mathsf{wf} \\ \Gamma' \vdash \overline{F}\ \mathsf{ok\ in}\ C \qquad \Gamma', \mathtt{this} : C\langle\overline{\beta}\rangle[\overline{\alpha}] \vdash \overline{M}\ \mathsf{ok\ in}\ C\langle\overline{\beta}\rangle[\overline{\alpha}] \qquad \Gamma' \vdash \overline{N}\ \mathsf{ok} \\ \Gamma' \vdash I\ \mathsf{ok\ in}\ C\langle\overline{\beta}\rangle[\overline{\alpha}] \qquad \Gamma' \vdash \overline{R}\ \mathsf{ok\ in}\ C \qquad \overline{M} \textit{overrides all methods with } \mathtt{this}_{\mathsf{fr}} \textit{ perm in C'}\end{array}}{\Gamma \vdash \mathtt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \mathtt{extends}\ C'\langle\overline{T}\rangle[\overline{a}]\{\overline{F}\ \overline{R}\ I\ \overline{N}\ \overline{M}\}\ \mathsf{ok}}$$

P-METHOD

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \overline{\alpha:\kappa}, \overline{x:T} \qquad \Gamma' \vdash \overline{\kappa\ \mathsf{wf}} \qquad \Gamma' \vdash \overline{T\ \mathsf{wf}} \qquad \Gamma' \vdash P\ \mathsf{wf} \\ \Gamma', \textit{result} : T_r \vdash P_r\ \mathsf{wf} \qquad \Gamma' \vdash \mathsf{override}(m, C\langle\overline{\beta}\rangle[\overline{\alpha_c}], \overline{x:T}, P \multimap \exists \textit{result} : T_r.P_r) \\ \Gamma'; P; \bullet \vdash_C e : \exists \textit{result} : T_r.P_r \otimes \top; \bullet\end{array}}{\Gamma \vdash m[\overline{\alpha:\kappa}](\overline{T\ x}) : P \multimap \exists \textit{result} : T_r.P_r = e\ \mathsf{ok\ in}\ C\langle\overline{\beta}\rangle[\overline{\alpha_c}]}$$

Figure 12: Well-formedness rules for the entire program.

4 Syntactic Sugar

Up until this point we have been assuming that programmer would be writing out the full specifications as we have presented them in our language. This system is quite flexible and expressive. However, given the syntactic complexity of some of the quantification bounds, for example $[\alpha_n : \mathsf{Node}_\beta, \alpha_\omega : \Omega(\mathsf{FF}(\alpha_n,\mathsf{alive},\beta), \mathsf{FF}(\alpha_n,\mathsf{alive},\beta)), \alpha_\xi : \Xi(\mathsf{LessThan1}), \alpha_A : \mathsf{Asmp}(\alpha_n,\alpha_\xi,\beta)]$ from our linked list example, we would really like to simplify things a bit! In this section we will introduce syntactic sugar that greatly simplifies our system of polymorphic access permissions while still retaining most of the expressiveness.

In order to simplify our system, we will introduce polymorphic variables that stand for entire Access Permissions, rather than for each permission element. These variables, when introduced, will be declared with one of three types of bounds:

Exact This variable bound introduces a permission that refers to a specific fractional quantity. Every time it is used, the instantiated permission will be required to be exactly the same.

Similar This variable bound introduces what is essentially a family of permissions each of the same permission kind. Every time this permission variable is used, instantiations are required to be of the same kind, but not necessarily the same fraction.

Symmetric This variable bound introduces a permission variable that is identical to 'Similar' in every way, and additionally can be divided an infinite number of times. Therefore, it can only be instantiated with permissions of kind pure and share (and immutable in our implementation).

Using these simplified bounds, the linked list class presented in the previous section could be written in the following manner:

```
class LinkedList<β>[p : symmetric(β)] {
```

$$\textsc{Var-}\Omega\ \frac{\Gamma \vdash \alpha : \Omega(\omega_1, \omega_2)}{\Gamma \vdash \alpha\ \mathsf{wf}} \qquad \textsc{Var-}\Xi\ \frac{\Gamma \vdash \alpha : \Xi(\xi)}{\Gamma \vdash \alpha\ \mathsf{wf}} \qquad \frac{\Gamma \vdash n_1 \leq n_2\ \mathsf{in}\ T \qquad \Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{FF}(n_1, n_2, T)\ \mathsf{wf}}$$

$$\frac{\Gamma \vdash n_1 \leq n_2\ \mathsf{in}\ T \qquad \Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{UFF}(n_1, n_2, T)\ \mathsf{wf}} \qquad \textsc{FF Type}\ \frac{\Gamma \vdash \omega_1\ \mathsf{wf} \qquad \Gamma \vdash \omega_2\ \mathsf{wf} \qquad \Gamma \vdash \omega_1 \sqsubseteq \omega_2}{\Gamma \vdash \Omega(\omega_1, \omega_2)\ \mathsf{wf}}$$

$$\textsc{Node}\ \frac{\Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{Node}_T\ \mathsf{wf}} \qquad \textsc{Assumption}\ \frac{\Gamma \vdash n : \mathsf{Node}_T \qquad \Gamma \vdash \kappa\ \mathsf{wf} \qquad \Gamma \vdash \kappa \sqsubseteq \mathsf{Fract} \Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{Asmp}(n, \kappa, T)\ \mathsf{wf}}$$

$$\textsc{Fracts}\ \Gamma \vdash \xi\ \mathsf{wf} \qquad \textsc{Fract Type}\ \Gamma \vdash \Xi(\xi)\ \mathsf{wf}$$

Figure 13: Rules for checking the well-formedness of quantification classifiers.

```
class Node {
  β item; Node next;
  alive = unique(next) ⊗ p(item)

  β get(int i, int cur):unique(this) ⊸ unique(this) ⊗ p(result)
}

alive = unique(first)
...
void add(β item) : unique(this) ⊗ p(item) ⊸ unique(this)

β get(int i) : unique(this) ⊸ unique(this) ⊗ p(result)
}
```

The permission variable p stands for a permission that can be divided any number of time but will still result in a permission of the same kind. Specifically, each time p is mentioned, it may refer to different fractions in the below fraction and the fraction function. Note that the bound of p must still declare the type β with which its permissions will be associated.

These new permission variables are truly syntactic sugar. They can be defined in terms of our lower level quantification variables. For each of the three types of bounds for permission variables, there is a different way to translate its declaration and its use. The table in Figure 14 summarizes the transformation from syntactic sugar to the formal language.

Of particular note is the translation of the use of a similar or symmetric permission variable. Each use is translated into an Access Permission that existentially quantifies the fraction and fraction functions. The classifiers of these existentially quantified variables are the classifiers introduced when the permission variable itself was declared. Additionally, the symmetric permission variable is rewritten as a series of quantification variables with a fraction classifier α_ξ, that is

Declaration and Use	
Sugar	**Rewrite**
$p : \mathsf{exact}(T)$	$\alpha_\xi : \Xi(\mathsf{Fract}), \alpha_n : \mathsf{Node}_T, \alpha_g : \mathsf{FF}(\alpha_n, \mathsf{alive}, T), \alpha_k : \alpha_\xi,$ $\alpha_A : \mathsf{Asmp}(\alpha_n, \alpha_\xi, T)$
$p(r)$	$\mathsf{access}(r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$
$p : \mathsf{similar}(T)$	$\alpha_n : \mathsf{Node}_T, \alpha_\omega : \Omega(\mathsf{UFF}(\alpha_n, \mathsf{alive}, T), \mathsf{FF}(\alpha_n, \mathsf{alive}, T)),$ $\alpha_\xi : \Xi(\mathsf{Fract}), \alpha_A : \mathsf{Asmp}(\alpha_n, \alpha_\xi, T)$
$p(r)$	$\exists\alpha_g{:}\alpha_\omega.\exists\alpha_k{:}\alpha_\xi.\mathsf{access}(r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$
$p : \mathsf{symmetric}(T)$	$\alpha_n : \mathsf{Node}_T, \alpha_\omega : \Omega(\mathsf{FF}(\alpha_n, \mathsf{alive}, T), \mathsf{FF}(\alpha_n, \mathsf{alive}, T)),$ $\alpha_\xi : \Xi(\mathsf{LessThan1}), \alpha_A : \mathsf{Asmp}(\alpha_n, \alpha_\xi, T)$
$p(r)$	$\exists\alpha_g{:}\alpha_\omega.\exists\alpha_k{:}\alpha_\xi.\mathsf{access}(r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$

Figure 14: The translation of permission variables, which are syntactic sugar, into the formal language, at both their declaration and use site.

bounded above by LessThan1, and a fraction function classifier α_ω, that is bounded below by FF.

Given such a large difference in syntactic complexity, readers may reasonably wonder whether or not our formal system could have been written to include these simplified polymorphic permissions from the start. Our motivation for presenting polymorphic access permissions in the manner is two-fold. First, we feel strongly that presenting the simplified polymorphic permissions in terms of a formal system where each element of the Access Permission can be quantified helps in understanding the semantics of the simplified permission bounds. This is particularly true for appreciating the different between the exact permission and the similar and symmetric permissions. It is crucial to understand that there is some extra level of quantification that is occurring in the later case that is not occurring in the former case. Second, the full system does allow some specifications that cannot be written in syntactic sugar. For example, if desired, a programmer could force multiple permissions to share the same guaranteed state. Still, due to the large gain in simplicity, we have chosen to implement the simplified syntax directly in our static analysis, described in the next section.

5 Implementation

In order to better evaluate polymorphic Access Permissions, we have implemented a typestate checker for the Java language based on this approach. Our implementation is an extension to the Plural [3] typestate checker for Java, which was in turn based on the original type system presented by Bierhoff and Aldrich [2]. Our polymorphic typestate checker implements the simplified system from the previous section directly, and does not allow abstract over each permission element. All of the specifications are written using Java 1.5 annotations. This presented a few interesting challenges.

The entire Plural implementation, which includes the polymorphic variant described here, the

source, and a suite of tests, is available for download[3]. This suite of tests includes Java versions of all of the examples presented in this paper, which are correctly verified.

The following listing is a specification of the `Node` class from our earlier linked list example, and serves to illustrate the basic form of the Java annotations that can legally be used in our implementation:

```
@Symmetric(value="p",type="T")
@Invariants(@State(name="alive",inv="unique(next) * p(item)"))
class Node<T> {
  @Apply("p") Node next; T item;

  @Unique
  @ResultPolyVar("p")
  T get(int i, int cur) {...}
}
```

The `@Symmetric` annotation introduces a polymorphic permission variable for the scope of the class, which must be associated with a type. The `@Exact` and `@Similar` annotations exist as well, and the permissions introduced have the same semantics presented in Section 4. The `@Invariants` and `@State` annotations are already a part of the Plural typestate checker, and are used to specify state invariants, but now polymorphic permissions can be used in these invariants. The `@ResultPolyVar` annotation, along with the `@PolyVar` annotation, allows us to mention these polymorphic permissions in specifications. Here is how we might instantiate a similarly specified `LinkedList` class:

```
@ResultShare("Open") Socket
getItemFromList(@Unique @Apply("share(Open)") LinkedList<Socket> l) {
  return l.get(0);
}
```

The `@Apply` annotation applies the share permission kind with a state guarantee of Open to the polymorphic permission parameter of `LinkedList`. At each application site, the applied permission is checked to ensure that it matches the bound on the parameter. Here, since the permission is share, it does. This permission kind and guarantee is subsequently substituted for p in the specification of the `get` method, and the result is that the post-condition of `getItemFromList` is satisfied. In this case, that means that `getItemFromList` returns a share permission with a guarantee of Open.

In our current implementation, polymorphic permissions can only be introduced at the class level, not at the method level. Polymorphic permissions must be instantiated at construction time. However, Java 1.5 annotations can not be used on constructor expressions. Therefore a very simple unification algorithm tracks the permissions that are applied to any expression.

Most of the checking functionality piggy-backs on top of the existing Plural tool. Within the scope of a polymorphic variable, a simple flow-based analysis tracks polymorphic permissions as they flow from specification to specification. This analysis treats polymorphic permission variables as being indivisible unless declared as symmetric. As previously mentioned, the analysis also tracks the instantiation of each reference. At method pre- and post-conditions, and receiver

[3] `http://code.google.com/p/pluralism/`

pack and unpack sites, this instantiation information is used to determine which permissions are consumed and which permissions are produced. In the case where a polymorphic permission is instantiated with an actual permission, our analysis substitutes the actual permission for the variable in the method specification, and then the original Plural implementation tracks whether or not the appropriate permissions are available in order to satisfy the method pre-condition, and also tracks the newly produced permissions.

6 Related Work

Existing approaches have contained some similar ideas to the ones presented here, particularly with respect to quantification. In the end, the novelty of our work comes from the manner in which these ideas have been combined, and the novel quantification bounds that we have used to extend modular typestate checking to generic classes.

The original type system upon which this work was based [2] contains a very limited form of quantification. This system allows existential and universal quantification over fractions and fraction functions, but only within the scope of the predicate syntactic form, P. This quantification was limited in many ways. Notably, the scope of the quantifiers could not extend over an entire method specification, only within a pre- or post-condition. Our work significantly improves upon the usefulness of the original approach by extending the scope of polymorphism to the method and class level, by allowing state guarantees and assumptions to be abstracted over, and by allowing quantification classifiers themselves to be abstracted over. This last point is what truly enabled the specification and verification of collections that we have seen in practice.

Boyland's fractional permissions [6] do allow polymorphism, by allowing universal quantification over fractions in procedure specifications. This allows programmers to write procedures that return the same fractions they were given, as long as the procedure body does not depend on them. The main difference is that our work supports a larger number of permission kinds (Boyland's work essentially supports unique and immutable) which means that we must support more interesting sorts of quantification. For instance, Boyland's work does not have an analogous notion of polymorphism over fraction and fraction function classifiers, likely because there are not enough permission kinds to make this a useful feature.

Higher-Order Separation Logic [4] is able to verify some similar sorts of behavioral properties as our work. For example, using standard logical quantifiers, a function can be defined that is polymorphic in the state of the objects that it accepts and returns. However, existing work does not allow polymorphism over the permission to heap locations. This is not surprising considering that most formulations of Separation Logic have only one "permission." That being said, recent work has extended fractional permissions to separation logic [5]. This work does not permit quantification over fractions themselves.

Finally, Girard's original work on Linear Logic [9] allowed for quantification over linear facts. However, this work was not presented in the context of managing program resources and therefore it is not clear how this quantification would translate to permission accounting for polymorphic programs.

7 Conclusion

In this paper we extended an existing type system, designed to prevent the misuse of object protocols, to allow for polymorphism over Access Permissions, the static predicates that track what state each object is in, and how those objects may be aliased. This results in increased precision in the specification of classes whose implementations do not constrain the elements they contain, such as a stack that is equally capable of holding unique, open files as shared, open sockets. Our experience has shown that this expressiveness is necessary in order to be able to specify commonly used classes without false positives. While this system was expressed in terms of a low level calculus where each part of an Access Permission can be abstracted individually, we showed a simplified syntax of our system that can be rewritten in terms of the underlying calculus and presented a typestate checker for Java based on this system.

References

[1] Kevin Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, Carnegie Mellon University, April 2009.

[2] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *The 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 301–320. ACM Press, 2007.

[3] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09)*, pages 195–0219, July 2009.

[4] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5):24, 2007.

[5] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

[6] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[7] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. *SIGPLAN Not.*, 36(5):59–69, 2001.

[8] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP '04: European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

[9] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

[10] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[11] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded java programs. *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 288–296, Sept. 2008.

[12] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *The 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 144–153. ACM Press, 1998.

[13] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.